No Start Menu for You

Posted on January 17, 2023 by brucedawson

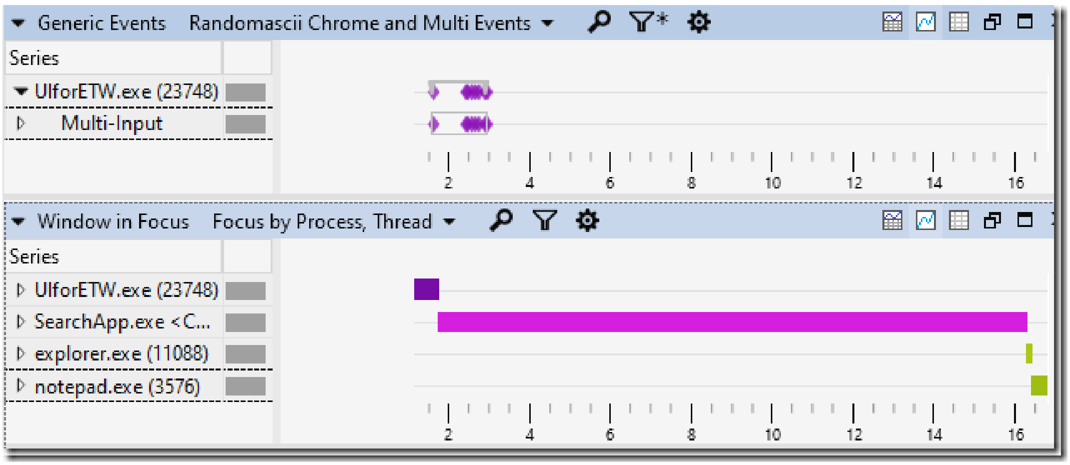
I tend to launch most programs on my Windows 10 laptop by typing the <Win> key, then a few letters of the program name, and then hitting enter. On my powerful laptop (SSD and 32 GB of RAM) this process usually takes as long as it takes me to type these characters, just a fraction of a second.

Usually.

Sometimes, however, it takes longer. A lot longer. As in, tens of seconds. The slowdowns are unpredictable but recently I was able to record an Event Tracing for Windows (ETW) trace of one of these delays. With a bit of help from people on twitter I was able to analyze the trace and understand why it took about a minute to launch *notepad*.

Before I get in to the analysis I have two warnings/disclaimers: 1) I have a good understanding of the problem, but I do not have a solution and 2) if you are seeing identical symptoms that doesn't mean that your root cause is the same as mine, but I will give some hints on how to see if it is.

My analysis of the trace (trace is here, installer for the analysis tools is here, analysis tutorials are here, feel free to follow along) started with my looking at the input events and *Window in Focus* graph in Windows Performance Analyzer (WPA), both shown below (zoomed in a bit for maximum detail):



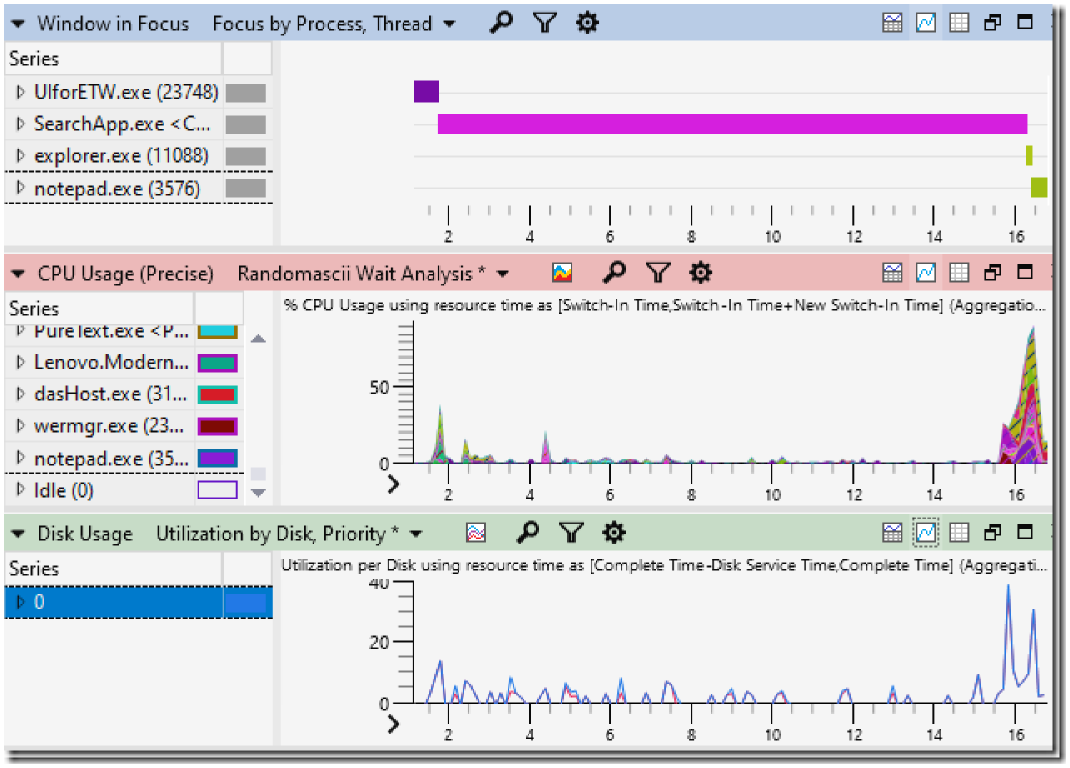
The first diamond in the *Multi-Input* row shows when I pressed the Windows key, with subsequent key presses (including pressing enter) clumped together shortly afterwards. The units on the x-axis are seconds so we can see that all of the typing took about 3/4 of a second.

The input events are injected into the trace by my UIforETW trace-recording tool. These input events are one of the reasons I prefer UIforETW over Microsoft's trace-recording tools. I've hidden the table-view but it lists which keys were pressed, while anonymizing letters and numbers to prevent UIforETW from being a key logger. Input events can be a critical tool in helping know where/when to look in traces to understand what is happening.

The input events help establish context, but the *Window in Focus* events really tell the tale. We can see that the SearchApp (start menu?) gains focus as soon as I press the <Win> key but then nothing else happens for more than thirteen seconds. That's the problem, visualized.

But why?

The next step is to see what is causing the delay. A quick glance at the CPU Usage (Precise) and Disk Usage graphs showed that the CPU and disk were almost 100% idle, so the start menu must be waiting on something else:

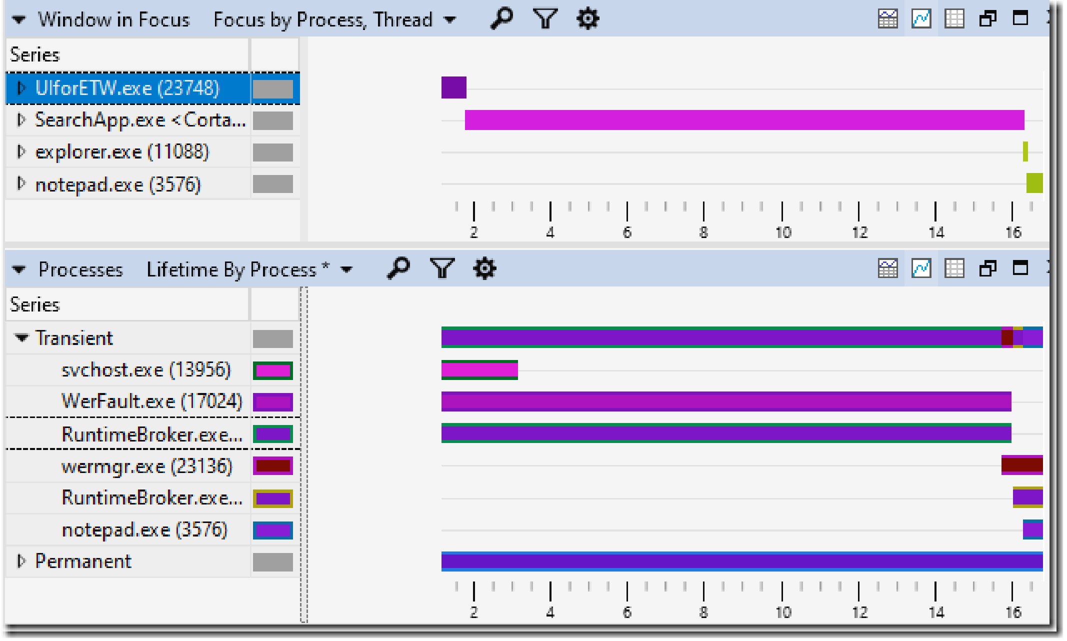

Window in Focus
Focus by Process, Thread
Series
UIforETW.exe (23748)
SearchApp.exe <C...
explorer.exe (11088)
notepad.exe (3576)
CPU Usage (Precise)
Randomascii Wait Analysis *
% CPU Usage using resource time as [Switch-In Time,Switch-In Time+New Switch-In Time] (Aggregatio...
PureText.exe <P...
Lenovo.Modern...
dasHost.exe (31...
wermgr.exe (23...
notepad.exe (35...
Idle (0)
Disk Usage
Utilization by Disk, Priority *
Utilization per Disk using resource time as [Complete Time-Disk Service Time,Complete Time] (Aggregati...
0

When a process is idle for a while when you wish that it was doing work then the challenge is to figure out what it was waiting on. I looked at the context-switch events in CPU Usage (Precise). Some of the *SearchApp* threads were named (yay!) but not all of them were and I couldn't find the main thread to see what it was waiting on, so I had to poke around and hope something became obvious. I zoomed in on the burst of CPU activity just before notepad launched and I noticed that *WerFault.exe* and *wermgr.exe* both started getting busy. Correlation is not causation, but it sure is suspicious.

> *Note that WER stands for Windows Error Reporting – the system that sends crash dumps back to Microsoft for analysis so that software reliability can be improved*

Looking at the *Processes* table showed me that the command line for *WerFault.exe* was "*C:\WINDOWS\system32\WerFault.exe -u -p 17804 -s 2124*". That suggests that Windows Error Reporting was being asked to record information for crashed process 17804, and when I looked in the *Processes* table for that Process ID (PID) I found "*RuntimeBroker.exe <Microsoft.Windows.Search> (17804)*". Well now. Doesn't that name look relevant?

A look at all of the "Transient" processes (those that started or ended during the trace timeline) was quite revealing:



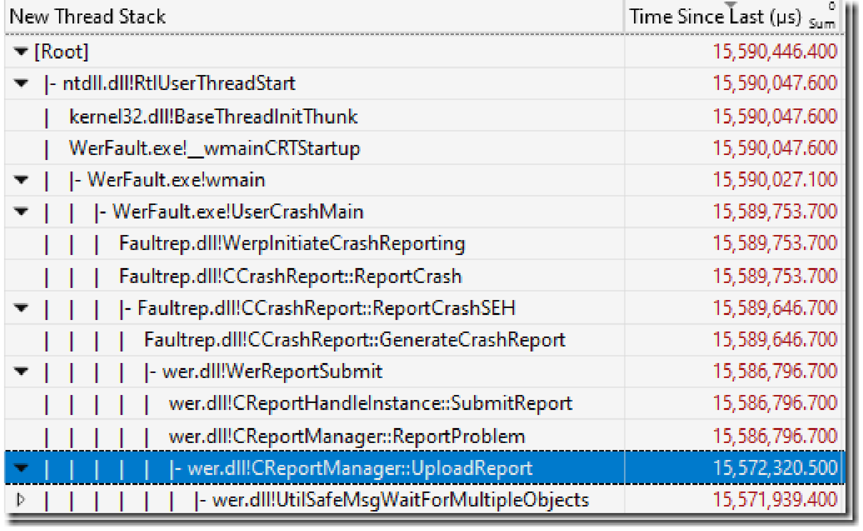
WerFault.exe and *RuntimeBroker.exe (17804)* (the top of the two *RuntimeBroker.exe processes*) were both running when I started recording the trace and both ended at about the same time, and *WerFault.exe* was handling a crash in *RuntimeBroker.exe*. Notice also that a new copy of *RuntimeBroker.exe* starts running when the old copy goes away. Now we're starting to have an explanation:

1. *RuntimeBroker.exe* crashes
2. WerFault.exe deals with the crash, keeping the *RuntimeBroker.exe* process open
3. Then a new *RuntimeBroker.exe* launches and provides whatever it is that *SearchApp.exe* needed

Now we have a new question: why is *WerFault.exe* sitting idle for so long?

I looked at the CPU Usage (Precise) data and saw that *WerFault.exe* has at least thirteen threads, none of them named (come *on* Microsoft – thread names are really helpful!) but the main thread was easily identifiable as the one using the most CPU time. I then sorted by *Time Since Last* and noticed that at one point the main thread had been waiting to run for 15.572 s. In fact it was probably waiting even longer, but the start of its wait was before the start of the trace and therefore unknowable. You can find more details on how to do idle-analysis here.

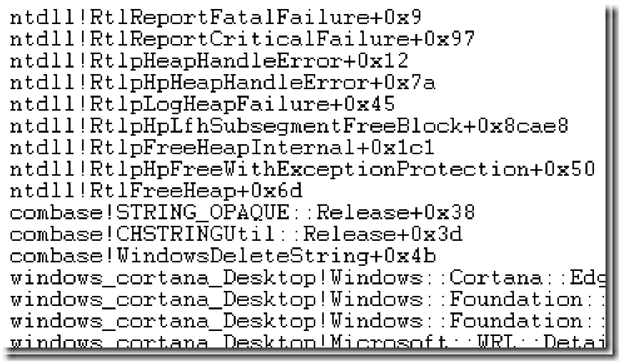
The stack where the main *WerFault.exe* thread was waiting for 15.572 s is shown below:



```
New Thread Stack                                              Time Since Last (µs) Sum
[Root]                                                        15,590,446.400
|- ntdll.dll!RtlUserThreadStart                               15,590,047.600
|  kernel32.dll!BaseThreadInitThunk                           15,590,047.600
|  WerFault.exe!__wmainCRTStartup                             15,590,047.600
|  |- WerFault.exe!wmain                                      15,590,027.100
|  |  |- WerFault.exe!UserCrashMain                           15,589,753.700
|  |  |  Faultrep.dll!WerpInitiateCrashReporting              15,589,753.700
|  |  |  Faultrep.dll!CCrashReport::ReportCrash               15,589,753.700
|  |  |  |- Faultrep.dll!CCrashReport::ReportCrashSEH         15,589,646.700
|  |  |  |  Faultrep.dll!CCrashReport::GenerateCrashReport    15,589,646.700
|  |  |  |  |- wer.dll!WerReportSubmit                       15,586,796.700
|  |  |  |  |  wer.dll!CReportHandleInstance::SubmitReport    15,586,796.700
|  |  |  |  |  wer.dll!CReportManager::ReportProblem          15,586,796.700
|  |  |  |  |  |- wer.dll!CReportManager::UploadReport        15,572,320.500
|  |  |  |  |  |  |- wer.dll!UtilSafeMsgWaitForMultipleObjects 15,571,939.400
```

The summary would be that it was waiting in *UploadReport*.

So now we understand the problem. *RuntimeBroker.exe* crashed (due to heap corruption, according to the call stack in the *RuntimeBroker.exe* crash dump, shown to the right) and it took more than 15 seconds to upload the crash dump, presumably due to my flaky hotel WiFi. During this time my start menu was inoperable.

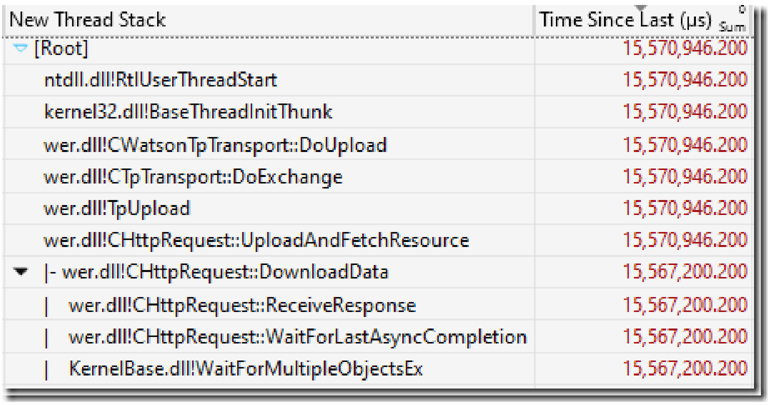


```
ntdll!RtlReportFatalFailure+0x9
ntdll!RtlReportCriticalFailure+0x97
ntdll!RtlpHeapHandleError+0x12
ntdll!RtlpHpHeapHandleError+0x7a
ntdll!RtlpLogHeapFailure+0x45
ntdll!RtlpHpLfhSubsegmentFreeBlock+0x8cae8
ntdll!RtlpFreeHeapInternal+0x1c1
ntdll!RtlpHpFreeWithExceptionProtection+0x50
ntdll!RtlFreeHeap+0x6d
combase!STRING_OPAQUE::Release+0x38
combase!CHSTRINGUtil::Release+0x3d
combase!WindowsDeleteString+0x4b
windows_cortana_Desktop!Windows::Cortana::Edg
windows_cortana_Desktop!Windows::Foundation::
windows_cortana_Desktop!Windows::Foundation::
windows_cortana_Desktop!Microsoft::WRL::Detai
```

This deserves reiterating. My start menu was hung due to the combination of heap corruption and *WerFault.exe* deciding that it needed to upload the crash dump before releasing the old process so that a new one could be started.

It took two bugs (the heap corruption and the upload-before-restarting) to make this hang happen, but happen it did.

We can even go deeper. The *UploadReport* function was blocked for 15.567 s and the *Readying Process/Readying Thread Id* shows us who ultimately unblocked the function. That turned out to be another *WerFault.exe* thread which was blocked in some *CHttpRequest* functions, as show above. That doesn't add significantly to the understanding of the problem, but does demonstrate nicely how you can trace a hang backwards through multiple processes and threads.



```
New Thread Stack                                      Time Since Last (µs) Sum
[Root]                                                15,570,946.200
ntdll.dll!RtlUserThreadStart                          15,570,946.200
kernel32.dll!BaseThreadInitThunk                      15,570,946.200
wer.dll!CWatsonTpTransport::DoUpload                  15,570,946.200
wer.dll!CTpTransport::DoExchange                      15,570,946.200
wer.dll!TpUpload                                      15,570,946.200
wer.dll!CHttpRequest::UploadAndFetchResource          15,570,946.200
|- wer.dll!CHttpRequest::DownloadData                 15,567,200.200
|  wer.dll!CHttpRequest::ReceiveResponse              15,567,200.200
|  wer.dll!CHttpRequest::WaitForLastAsyncCompletion   15,567,200.200
|  KernelBase.dll!WaitForMultipleObjectsEx            15,567,200.200
```

Watching for this problem

In general if you want to understand why your computer is performing badly you need to record and analyze a trace. However if you want to see if you are hitting this particular problem then there are easier steps that you can follow.

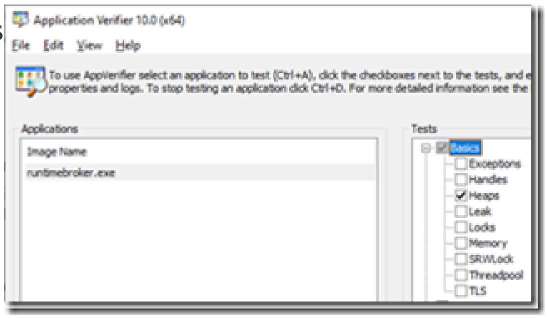
The first step is to configure the local recording of crash dumps. This is a good idea in general because it lets you monitor the stability of your computer over your time.

Then, with crash dumps being recorded if you see a Start Menu hang you can just look in %localappdata%\crashdumps and see if there is a recent *RuntimeBroker.exe* crash. If so then you are presumably seeing this bug.

Waiting on upload

Raymond Chen gives several reasons for [why Windows Error Reporting doesn't restart crashed processes before uploading the report]() (circa 2012) but I don't find those reasons entirely compelling, especially in the start-menu case. As long as you kill the old process before starting the new one – and answer the DLL-version questions from a crash dump – most of the problems he points out are avoidable. Exponential backoff on process restarts can address the rest. And, the consequences of waiting can, as we have seen, be arbitrarily long start-menu hangs, with no indication that a crash is the problem. There is also [some confusion about the behavior]() – maybe the design has changed in the last ten years.

Fixing the crash

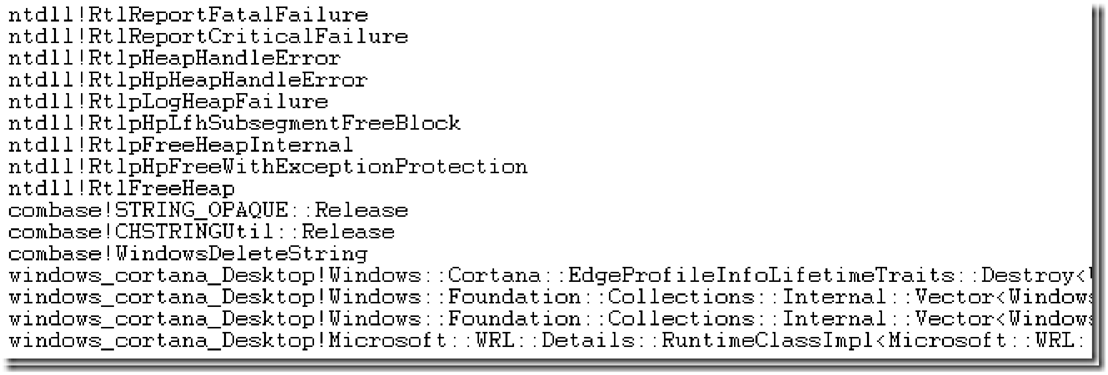


To use AppVerifier select an application to test (Ctrl+A), click the checkboxes next to the tests, and e
properties and logs. To stop testing an application click Ctrl+D. For more detailed information see the

runtimebroker.exe

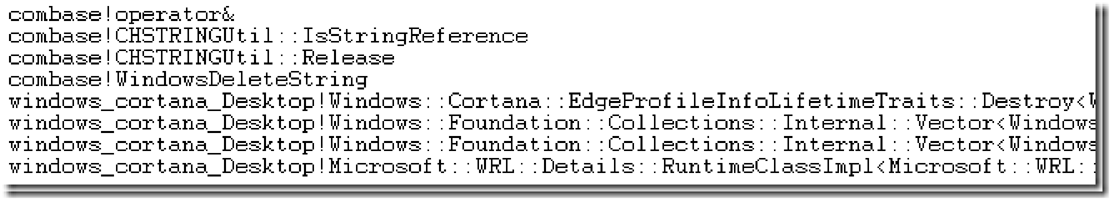
Heap corruption bugs can be extremely difficult to find and fix, but this one seems like it might be easy. I turned on [pageheap]() on *RuntimeBroker.exe*, killed the relevant version to get it to restart and apply the *pageheap* settings and it started crashing every time I opened the start menu. I configured WER to save full crash dumps and soon had a half-dozen crash dumps with full details of what was happening.

The crashes normally happen on this call stack:



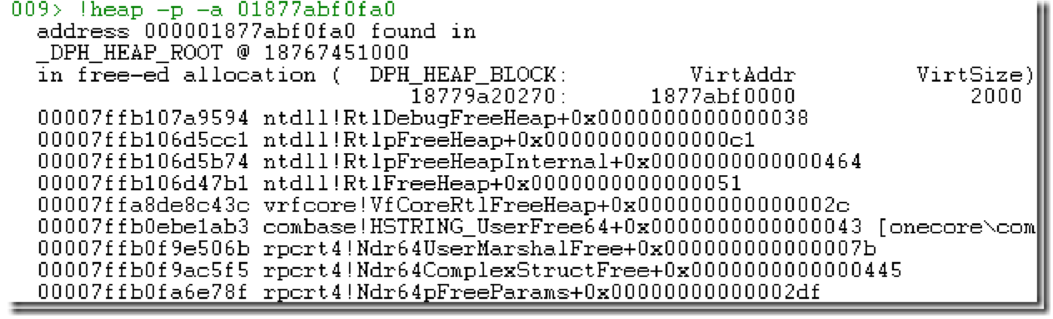
```
ntdll!RtlReportFatalFailure
ntdll!RtlReportCriticalFailure
ntdll!RtlpHeapHandleError
ntdll!RtlpHpHeapHandleError
ntdll!RtlpLogHeapFailure
ntdll!RtlpHpLfhSubsegmentFreeBlock
ntdll!RtlpFreeHeapInternal
ntdll!RtlpHpFreeWithExceptionProtection
ntdll!RtlFreeHeap
combase!STRING_OPAQUE::Release
combase!CHSTRINGUtil::Release
combase!WindowsDeleteString
windows_cortana_Desktop!Windows::Cortana::EdgeProfileInfoLifetimeTraits::Destroy<W
windows_cortana_Desktop!Windows::Foundation::Collections::Internal::Vector<Window
windows_cortana_Desktop!Windows::Foundation::Collections::Internal::Vector<Windows
windows_cortana_Desktop!Microsoft::WRL::Details::RuntimeClassImpl<Microsoft::WRL:
```

With pageheap enabled the crash happens on a very similar call stack, but slightly earlier. The crash happens earlier (and more reliably) because with page heap when you free memory it is unmapped, so dereferencing it reliably causes a crash, instead of reading from the freed memory:



```
combase!operator&
combase!CHSTRINGUtil::IsStringReference
combase!CHSTRINGUtil::Release
combase!WindowsDeleteString
windows_cortana_Desktop!Windows::Cortana::EdgeProfileInfoLifetimeTraits::Destroy<W
windows_cortana_Desktop!Windows::Foundation::Collections::Internal::Vector<Windows
windows_cortana_Desktop!Windows::Foundation::Collections::Internal::Vector<Windows
windows_cortana_Desktop!Microsoft::WRL::Details::RuntimeClassImpl<Microsoft::WRL:
```

The crash happens when dereferencing *[rcx]* so I ran its value through the the !heap command (see my [pageheap blogpost]() for details) and got this call stack:



```
009> !heap -p -a 01877abf0fa0
    address 000001877abf0fa0 found in
    _DPH_HEAP_ROOT @ 18767451000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                 18779a20270:      1877abf0000             2000
    00007ffb107a9594 ntdll!RtlDebugFreeHeap+0x0000000000000038
    00007ffb106d5cc1 ntdll!RtlpFreeHeap+0x00000000000000c1
    00007ffb106d5b74 ntdll!RtlpFreeHeapInternal+0x0000000000000464
    00007ffb106d47b1 ntdll!RtlFreeHeap+0x0000000000000051
    00007ffa8de8c43c vrfcore!VfCoreRtlFreeHeap+0x000000000000002c
    00007ffb0ebe1ab3 combase!HSTRING_UserFree64+0x0000000000000043 [onecore\com
    00007ffb0f9e506b rpcrt4!Ndr64UserMarshalFree+0x000000000000007b
    00007ffb0f9ac5f5 rpcrt4!Ndr64ComplexStructFree+0x0000000000000445
    00007ffb0fa6e78f rpcrt4!Ndr64pFreeParams+0x00000000000002df
```

The only complication is that this doesn't happen on all Windows 10 machines. There seems to be some required state that makes it happen or not and I don't know what is. I will say that I'm happy to share the crash dumps with anyone at Microsoft who wants to investigate.

I don't know the code and don't understand what is happening but I've dealt with enough use-after-free bugs to say that this is probably straightforward to investigate and fix using the crash dumps. Although, I got a couple of different crash call stacks so there might be multiple bugs.

Conclusions

I ended my initial twitter thread by saying that these hangs were making me cranky and had me wondering if Windows 10 was abandonware. Since then I've been told that people seem to be seeing start-menu hangs on Windows 11, but that is not necessarily the same problem.

To be clear, Microsoft has the technology to record traces on start menu hangs on customer machines. These traces would show roughly the same thing as my trace. They also receive crash dumps from customer machines. They might even have a way of correlating them (if not then they should hook that up). And they created *pageheap* which makes use-after-free crashes easy to investigate.

So, why hasn't this been addressed? On my laptop I see that *RuntimeBroker.exe* has crashed, on average, every second day this year. That is too many start-menu hangs for my tastes. I don't know how long it has been happening so maybe a fix is on the way – if so that would be great to hear. If not then I will continue to be cranky and I hope that this serves as a good reminder of the importance of using all that fancy telemetry to address issues like this.

Or, maybe I'm just unlucky and I'm one of the few people (not the only person) who is hitting this crash.

In short, I am *really* pleased with the tools that Microsoft has created and released to let me analyze performance issues such as these. However I wish that I didn't have to use them so often on Windows itself.

Online discussion

Hacker news

Initial twitter discussion

Twitter announcement of blogpost

About brucedawson

I'm a programmer, working for Google, focusing on optimization and reliability. Nothing's more fun than making code run 10x as fast. Unless it's eliminating large numbers of bugs. I also unicycle. And play (ice) hockey. And sled hockey. And juggle. And worry about whether this blog should have been called randomutf-8. 2010s in review tells more: https://twitter.com/BruceDawson0xB/status/1212101533015298048

View all posts by brucedawson →

This entry was posted in Code Reliability, Debugging, Investigative Reporting, Performance, Programming, Rants, uiforetw, xperf and tagged hangs, pageheap, Windows 10 abandonware. Bookmark the permalink.

6 Responses to *No Start Menu for You*

djmips *says:*
January 17, 2023 at 11:20 pm

Funny – after reading I went to check my CrashDumps folder and I had 10 RuntimeBroker.exe dmp files with roughly the same call stack and all starting on 1/14 2023 with the most recent one from today at 11:06 PM (1/17 2023)

Reply

anreiz *says:*
January 17, 2023 at 11:47 pm

Crashdumps shows RuntimBroker.exe.dmp from 2 days ago & 8 days ago. Always helps the crankyness to better understand the issue so kudos!

Reply

ignacy130 *says:*
January 17, 2023 at 11:48 pm

Microsoft has a huge problem with anything related to search. Windows Search is unusable, but I've counted on the ability to quickly launch apps via win+r… for nothing. It's hard to believe that a quick text search is so hard and one need to use Everything or some 3rd party launchers.

Reply

kasper93 *says:*
January 18, 2023 at 12:23 am

I have just-in-time debugger enabled on my main machine. I don't like when something is crashing in the background without my knowledge. I would rather fix or disable offending party.

That said I've seen this RuntimeBroker.exe crash, but only few times, definitely nothing reproducible enough to care about. I've seen also WindowsSearch.exe crash some time ago, but this doesn't happened anymore.

Nice writeup, always getting to the bottom of things 🙂

Reply

Stuart Axon *says:*

January 18, 2023 at 1:56 am

Thank you for looking at this!

I use Win-Key+typing in GNOME on Linux to launch everything, but I’ve always found this really painful on windows for some reason – I can’t pin it down but it feels very janky.

Reply

randommarius *says:*

January 18, 2023 at 3:52 am

I had similar issues. Windows search also does too much phoning home in my opinion.

My solutions was to use their Power Toys version of Run. It’s much faster and more configurable.

Reply